Understanding ResNets: A Deep Dive into Residual Networks with PyTorch

In this article, we learn how—and why—ResNets work and discover how to build our own. We implement a ResNet model using PyTorch and PyTorch Image Models (TIMM).

Aman Arora

Last Updated: Oct 2, 2023

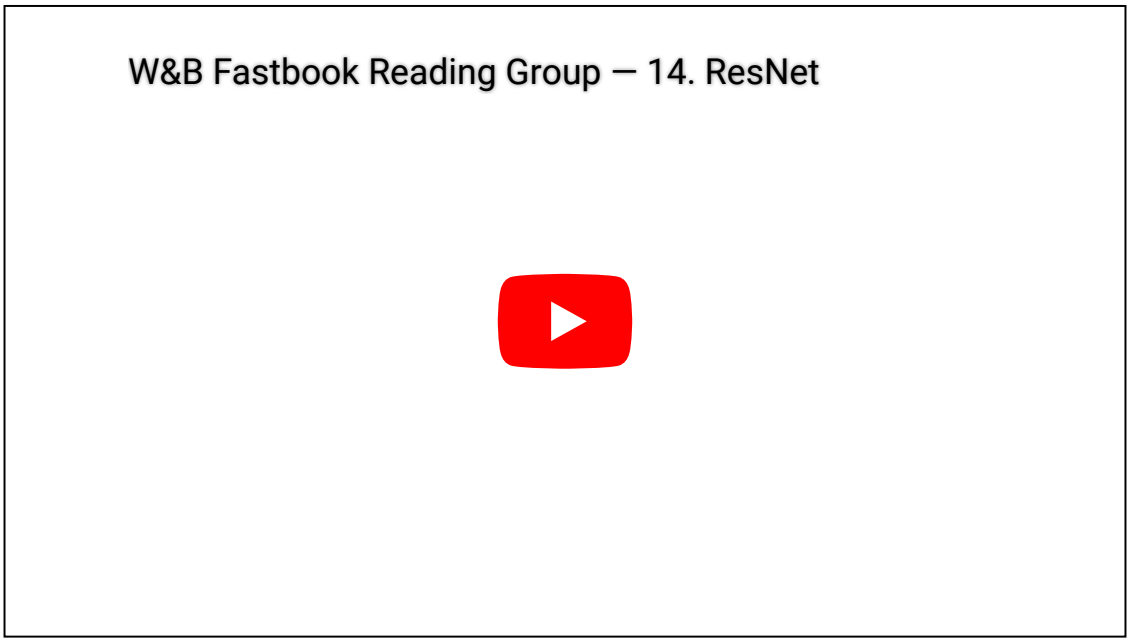
Deep learning has revolutionized the field of computer vision, enabling machines to recognize and classify images with human-like accuracy. One of the most influential architectures in this domain is the **Residual Network**, better known as ResNet.

Introduced by Kaiming He et al. in their 2015 paper, "**Deep Residual Learning for Image Recognition**," ResNets have since become a staple in the deep learning community, often serving as a starting point for many image classification tasks.

In this blog pos ResNets, understand wh ResNet model using **PyTorch** and **PyTorch Image Models (TIMM)**.

> 💡 This blog post is a blend of theory and practical implementation. We'll be using Python and PyTorch, so a basic understanding of these is recommended. We'll also be using TIMM, a library that provides pre-trained models and training scripts for PyTorch.

If you prefer learning about ResNets via video, please feel free to refer to my paper reading video on ResNets:



Here's what we'll be covering:

Table of Contents

Let's get going!

The Problem with Deep Networks

Before we dive into ResNets, let's understand the problem they were designed to solve. As we increase the depth of a neural network, we generally expect its performance to improve. But in 2015, the authors of the ResNet paper noticed something that they found curious. Even after using batchnorm, they saw that a network using more layers was doing less well than a network using fewer layers—and there were no other differences between the models.

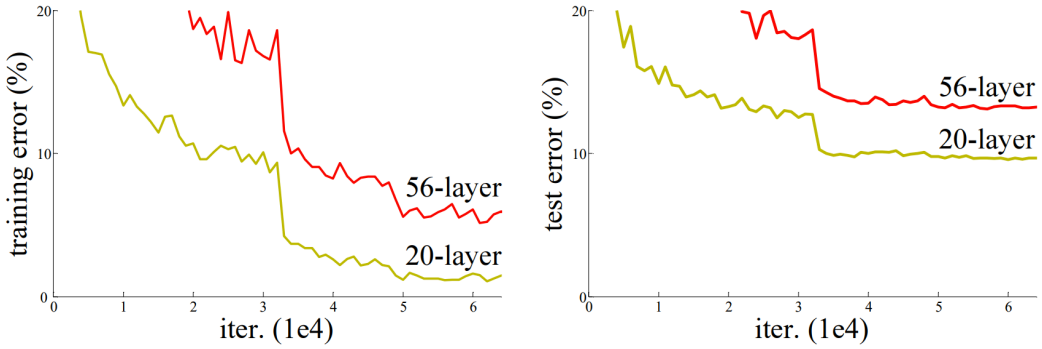
Most interestingly, the difference was observed not only in the validation set but also in the training set. In other words, this wasn't just a generalization issue but a training issue.

From the source:

> Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error, as [previously reported] and thoroughly verified by our experiments.

In the academic paper, this process is described in a rather inaccessible way, but the concept is actually very simple: start with a 20-layer neural network that is trained well, and add another 36 layers that do nothing at all (for instance, they could be linear layers). The result will be a 56-layer network that does exactly the same thing as the 20-layer network, proving that there are always deep networks that should be *at least as good* as any shallow network. But for some reason, stochastic gradient descent (SGD) does not seem able to find them.

This degradation problem wasn't due to overfitting but rather the difficulty of optimizing very deep networks. This is referred to as the **vanishing gradient problem.**



Also from the paper:

> When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep **model leads to higher training error**, verified by our experiments.

As can be seen in the figure above, not only is the test error higher for 56-layer network, but also the training error is higher. This meant that the degradation was not due to overfitting but rather due to complexity in the network.

When training neural networks with gradient-based learning methods and backpropagation, each of the neural network's weights receives an update proportional to the gradient of the error function with respect to the current weight in each iteration of training.

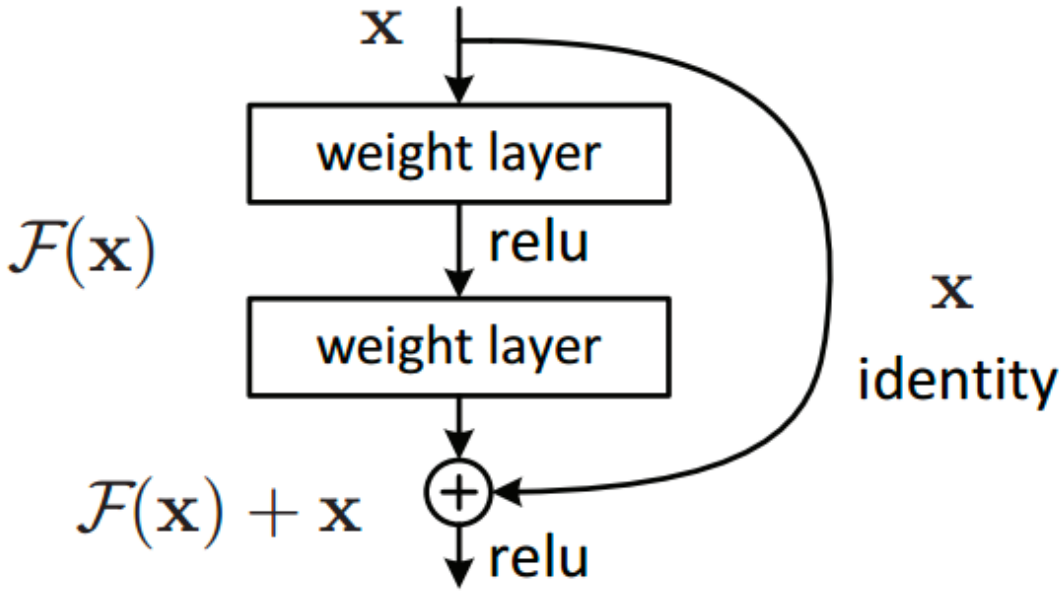
The problem is that, in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

As the backpropagation algorithm computes gradients using the chain rule, this has the effect of multiplying n of these small numbers to compute gra network, meaning that t nentially with n while the front layers train very slowly.

> 💡 A good experiment here for the reader will be to try to replicate figure above, and actually get worse results for deeper 56-layer model compared to 20-layer model. What do you think the chart would like for say 35-layers?

Enter ResNets

ResNets introduced a novel architecture to combat the vanishing gradient problem. The core idea of ResNet is introducing a so-called **identity shortcut connection** that skips one or more layers, as shown in the following figure:



The figure shows a building block of ResNet. Here, X is the input to the block and $F(X)$ represents the residual mapping to be learned. The operation $F(X) + X$ is performed by a shortcut connection and element-wise addition.

The idea of shortcut connections (or skip connections) allows the model to skip layers and helps to mitigate the problem of vanishing gradients. These connections allow the gradients to be directly backpropagated to earlier layers, which makes the network easier to optimize.

What have those skip connections gained us? The key thing is that those 36 extra layers, as they stand, are *identity mapping*, but they

have *parameters*, which means they are ***trainable***. So, when add the 36 extra layers, the model has the option to skip the layers, thus the new model is at least as good as the 20 layer model. Those extra 36 layers can then learn the parameters that make them most useful.

With the basic idea of ResNets now in place, let's move on to implementing ResNets first with TIMM and then from scratch using PyTorch.

Implementing ResNet with PyTorch and TIMM

First, we'll need to install the `timm` package. You can do this using pip:

```
bashCopy code
pip install timm
```

Next, we'll import the necessary libraries:

```
import torch
import timm
```

We can create a ResNet model using the `create_model` function from `timm`. Here, we're creating a ResNet50 model:

```
model = timm.create_model('resnet50', pretrained=True)
```

The `pretrained=True` argument means that we're using a model that has been pre-trained on the **ImageNet dataset**. This allows us to leverage the knowledge that the model has already gained from training on millions of images.

Now, let's see how we can use this model to make predictions. We'll start by creating a random tensor that will serve as our input:

```
x = torch.randn(1, 3, 224, 224)
```

This tensor rep ns are as follows: **1 image in the batch, 3 color channels (red, green, blue), and a height and width of 224 pixels.**

We can pass this tensor through our model to get the output:

```
output = model(x)
```

The output is a tensor of shape `(1, 1000)`, representing the probabilities of the image belonging to each of the 1000 classes in the ImageNet dataset.

Take a bow. **You've just implemented a ResNet model using TIMM.**

Creating ResNets from Scratch in Pure PyTorch

Now let's dive into the code and see how we can implement a ResNet from scratch in PyTorch.

Remember from the image that $F(X)$ is basically a convolution layer, followed by ReLU, followed by another convolution layer.

First, let's define the basic building block of a ResNet, the residual block:

```
import torch
from torch import nn

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size
                               stride=stride, padding=1, bias=False)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_siz
                               stride=1, padding=1, bias=False)

    def forward(self, x):
        out = self.conv2(self.relu(self.conv1(x)))
        out += x
        out =
```

As can be seen in the code above, it completely follows the figure. Here, $F(X)$ is defined as `self.conv2(self.relu(self.conv1(x)))`. Next, we just add X to $F(X)$ like so `out += x`. Finally, we take ReLU again and return the output.

That is really all it takes to implement a ResNet block from scratch in PyTorch!

But the above vanilla ResNet Block is not enough to implement the complete ResNet architecture. You might have seen variants of ResNet in the wild - resnet-34, resnet-50 or resnet-101 and so on. From the paper, the ResNet architecture variants are defined as in the following image.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

As can be seen from the architecture definitions above, we need to allow the model to go from 64 → 128 → 256 → 512 channels while decreasing the output size at the same time. Thus, we update our Vanilla ResNet block implementation as below.

```
import torch
from torch import nn

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size
                                                          =1, bias=False)
        self.
        self.relu = nn.ReLU(inplace=True)
```

```
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_siz
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, s
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
                               return out
```

This block consists of two convolutional layers, each followed by a batch normalization layer and a ReLU activation function. If the input and output dimensions do not match, we also add a shortcut connection that transforms the input to the required dimensions.

Now, we can define the full ResNet architecture. A ResNet is composed of several of these blocks stacked on top of each other. Here is a simple implementation of a ResNet:

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, paddi
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], str
        self.layer2 = self._make_layer(block, 128, num_blocks[1], st
        self.                                    _blocks[2], st
        self.                                    _blocks[3], st
        self.linear = nn.Linear(512, num_classes)
```

```
    def _make_layer(self, block, out_channels, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stri
            self.in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

In this code, the `_make_layer` function is used to create each layer of the network, which consists of several residual blocks with the same output size. The stride is set to 2 for the first block of each layer (except the first layer), which reduces the spatial dimensions of the output by half, effectively making it a downsampling layer.

The `forward` function defines the forward pass of the network. The input is passed through each layer in turn, and finally reshaped and passed through a fully connected layer to produce the output.

With this, we have a complete implementation of a ResNet in PyTorch! This model can be trained on a variety of tasks, including image classification, and has achieved state-of-the-art performance on many benchmarks.

Training the Basic Building Block in PyTorch

Now that we have defined our basic building block and the complete ResNet archite[illegible]e PyTorch library to train our ResNet on the ImageNette dataset.

To download the dataset, simply run the following commands.


```
tar -xvf imagenette2-160.tgz
```

Now you should have a `data` directory in the repository whose folder structure looks like:

```
data/
└── imagenette2-160
    ├── train
    │   ├── n01440764
    │   ├── n02102040
    │   ├── n02979186
    │   ├── n03000684
    │   ├── n03028079
    │   ├── n03394916
    │   ├── n03417042
    │   ├── n03425413
    │   ├── n03445777
    │   └── n03888257
    └── val
        ├── n01440764
        ├── n02102040
        ├── n02979186
        ├── n03000684
        ├── n03028079
        ├── n03394916
        ├── n03417042
        ├── n03425413
        ├── n03445777
        └── n03888257
```

Now that we have downloaded the data let's pre-process the dataset.

Preprocessing the Dataset

Before we can process our data. We will use the torchvision library to load the ImageNette dataset and apply the necessary transformations.

```
from torchvision import datasets, transforms

# Define transformations
transform = transforms.Compose(
    [transforms.Resize((224, 224)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Load ImageNette dataset
trainset = torchvision.datasets.ImageFolder(
        TRAIN_DATA_DIR, transform=transform
    )
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, sh

testset= torchvision.datasets.ImageFolder(
        TEST_DATA_DIR, transform=transform
    )
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuf
```

Now that our dataset is ready, we will create our model.

Creating the Model

Let's create resnet-34 and store it as `net`.

```
import torch.optim as optim

# Instantiate the network
net = ResNet(ResidualBlock, [3, 4, 6, 3])

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001, momentum=0.9)

# Move the network to GPU if available
if torch.cuda.is_available():
    net = net.cuda()
```

We also create

Training the Network

Now we can train our network. We will use W&B to log our training progress.

```
import wandb

# Initialize wandb
wandb.init(project="resnet-training")

# Training loop
for epoch in range(10):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        if torch.cuda.is_available():
            inputs = inputs.cuda()
            labels = labels.cuda()

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running
            running_loss = 0.0

        # Log the loss to wandb
        wandb.log({"loss": running_loss})

print('Finish
```

We iterate through out `tranloader` and pass the pre-processed images to ResNet-34 architecture to get model outputs. Next, we calculate the loss and perform backpropagation to update the architecture's parameters. We also log the loss to W&B.

Evaluating the Network

After training our network, we can evaluate its performance on the test set. We will also log the accuracy to W&B.

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        if torch.cuda.is_available():
            images = images.cuda()
            labels = labels.cuda()
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the test images: %d %%' % (100 * cc

# Log the accuracy to wandb
wandb.log({"accuracy": correct / total})
```

In the script above, we get the outputs from the resnet-34 archtiecture, and also get predicted labels. We could have also used `argmax` instead of `torch.max`. Finally, our accuracy is simply `correct / total`.

Conclusion

In this post, we have seen how to implement and train a ResNet architecture fr[illegible]r model on the ImageNette[illegible]We also used W&B to log our training progress and final accuracy. This shows the

power of ResNets and how they can be used for image classification tasks. With this knowledge, you can now experiment with different configurations and datasets to further improve your model's performance. Happy training!

Tags: Articles, ResNet, Intermediate, Domain Agnostic

Created with ❤️ on Weights & Biases.

https://wandb.ai/amanarora/Written-Reports/reports/Understanding-ResNets-A-Deep-Dive-into-Residual-Networks-with-PyTorch--Vmlldzo1MDAxMTk5
